Oblivious Neural Network Predictions via MiniONN transformations

Jian Liu
Aalto University
jian.liu@aalto.fi

Mika Juuti
Aalto University
mika.juuti@aalto.fi

Yao Lu
Aalto University
yao.lu@aalto.fi

N. Asokan
Aalto University
asokan@acm.org

ABSTRACT

Machine learning models hosted in a cloud service are increasingly popular but risk privacy: clients sending prediction requests to the service need to disclose potentially sensitive information. In this paper, we explore the problem of privacy-preserving predictions: after each prediction, the server learns nothing about clients' input and clients learn nothing about the model.

We present MiniONN, the first approach for *transforming an existing neural network* to an *oblivious neural network* supporting privacy-preserving predictions with reasonable efficiency. Unlike prior work, MiniONN requires *no change to how models are trained.* To this end, we design oblivious protocols for commonly used operations in neural network prediction models. We show that MiniONN outperforms existing work in terms of *response latency and message sizes.* We demonstrate the wide applicability of MiniONN by transforming several *typical* neural network models trained from *standard datasets.*

CCS CONCEPTS

• **Security and privacy → Privacy-preserving protocols**;

KEYWORDS

privacy, machine learning, neural network predictions

1 INTRODUCTION

Machine learning is now used extensively in many application domains such as pattern recognition [10], medical diagnosis [24] and credit-risk assessment [3]. Applications of *supervised* machine learning methods have a common two-phase paradigm: (1) a *training phase* in which a model is trained from some training data, and (2) a *prediction phase* in which the trained model is used to predict categories (classification) or continuous values (regression) given some input data. Recently, a particular machine learning framework, *neural networks* (sometimes referred to as *deep learning*), has gained much popularity due to its record-breaking performance in many tasks such as image classification [36], speech recognition [19] and complex board games [34].

Machine learning as a service (MLaaS) is a new service paradigm that uses cloud infrastructures to train models and offer online prediction services to clients. While cloud-based prediction services have clear benefits, they put clients' privacy at risk because the input data that clients submit to the cloud service may contain sensitive information. A naive solution is to have clients download the model and run the prediction phase on client-side. However, this solution has several drawbacks: (1) it becomes more difficult for service providers to update their models; (2) the trained model may constitute a competitive advantage and thus requires confidentiality; (3) for security applications (e.g., spam or malware detection services), an adversary can use the model as an oracle to develop strategies for evading detection; and (4) if the training data contains sensitive information (such as patient records from a hospital) revealing the model may compromise privacy of the training data or even violate regulations like the Health Insurance Portability and Accountability Act of 1996 (HIPAA).

A natural question to ask is, *given a model, whether is it possible to make it oblivious*: it can compute predictions in such a way that the server learns nothing about clients' input, and clients learn nothing about the model except the prediction results. For general machine learning models, nearly practical solutions have been proposed [6, 13, 14, 56]. However, privacy-preserving deep learning prediction models, which we call *oblivious neural networks* (ONN), have not been studied adequately. Gilad-Bachrach et al. [27] proposed using a specific activation function ("square") and pooling operation (mean pooling) during training so that the resulting model can be made oblivious using their CryptoNets framework. CryptoNets transformations result in reasonable accuracy but incur high performance overhead. Very recently, Mohassel and Zhang [43] also proposed new activation functions that can be efficiently computed by cryptographic techniques, and use them in the training phase of their SecureML framework. What is common to both approaches [27, 43] is that they require changes to the training phase and thus are not applicable to the problem of making *existing* neural models oblivious.

In this paper, we present MiniONN (pronounced minion), a practical ONN transformation technique to convert *any given neural network model* (trained with *commonly used operations*) to an ONN. We design oblivious protocols for operations routinely used by neural network designers: *linear transformations*, popular *activation functions* and *pooling operations.* In particular, we use *polynomial splines* to approximate nonlinear functions (e.g., sigmoid and tanh) with negligible loss in prediction accuracy. None of our protocols require any changes to the training phase of the model being transformed. We only use *lightweight* cryptographic primitives such as secret sharing and garbled circuits in online prediction phase. We also introduce an offline precomputation phase to perform request-independent operations using additively homomorphic encryption together with the SIMD batch processing technique.

Our contributions are summarized as follows:

- We present MiniONN, the *first* technique that can **transform any common neural network model into an oblivious neural network** without any modifications to the training phase (Section 4).
- We design **oblivious protocols for common operations in neural network predictions** (Section 5). In particular, we **make nonlinear functions (e.g., sigmoid and tanh) amenable for our ONN transformation** with a negligible loss in accuracy (Section 5.3.2).
- We build a **full implementation of MiniONN** and demonstrate its wide applicability by using it to transform neural network models **trained from several standard datasets** (Section 6). In particular, for the *same* models trained from the MNIST dataset [37], MiniONN performs **significantly better** than previous work [27, 43] (Section 6.1).
- We analyze how **model complexity** impacts both **prediction accuracy** and **computation/communication overhead** of the transformed ONN. We discuss how a neural network designer can choose the right tradeoff between prediction accuracy and overhead. (Section 7).

2 BACKGROUND AND PRELIMINARIES

We now introduce the machine learning and cryptographic preliminaries (notation we use is summarized in Table 1).

$\mathcal{S}$	Server
$\mathcal{C}$	Client
$\mathbf{X} = \{\mathbf{x}_1, \ldots\}$	Input matrix for each layer
$\mathbf{W} = \{\mathbf{w}_1, \ldots\}$	Weight matrix for each layer
$\mathbf{B} = \{\mathbf{b}_1, \ldots\}$	Bias matrix for each layer
$\mathbf{Y} = \{\mathbf{y}_1, \ldots\}$	Output matrix for each layer
$\mathbf{z} = \{z_1, \ldots\}$	Final predictions
u	$\mathcal{S}$'s share of the dot-product triple
v	$\mathcal{C}$'s share of the dot-product triple
$\mathbb{Z}_N$	Plaintext space
$compare(x, y)$	return 1 if $x \geq y$, return 0 if $x < y$
$E()$ / $D()$	Additively homomorphic encryption/decryption
pk / sk	Public/Private key
$\hat{x}$	$E(pk, x)$
$\widetilde{\mathbf{x}}$	$E(pk, [x_1, \ldots])$
$\oplus$	Addition between two ciphertexts or a plaintext and a ciphertext
$\ominus$	Subtraction between two ciphertexts or a plaintext and a ciphertext
$\otimes$	Multiplication between a plaintext and a ciphertext

Table 1: Notation table.

2.1 Neural networks

A neural network consists of a pipeline of layers. Each layer receives input and processes it to produce an output that serves as input to the next layer. Conventionally, layers are organized so that the bottom-most layer receives input data (e.g., an image or a word) and the top-most layer outputs the final predictions. A typical neural network[1] processes input data in groups of layers, by first applying *linear transformations*, followed by the application of a nonlinear *activation function*. Sometimes a *pooling operation* is included to aggregate groups of inputs.

We will now briefly describe these operations from the perspective of transforming neural networks to ONNs.

2.1.1 Linear transformations. The commonest linear transformations in neural networks are matrix multiplications and additions:

$$\mathbf{y} := \mathbf{W} \cdot \mathbf{x} + \mathbf{b}, \tag{1}$$

where $\mathbf{x} \in \mathbb{R}^{l \times 1}$ is the input vector, $\mathbf{y} \in \mathbb{R}^{n \times 1}$ is the output, $\mathbf{W} \in \mathbb{R}^{n \times l}$ is the *weight matrix* and $\mathbf{b} \in \mathbb{R}^{n \times 1}$ is the *bias vector*.

Convolution is a type of linear transformation, which computes the dot product of small "weight tensors" (filters) and the neighborhood of an element in the input. The process is repeated, by sliding each filter by a certain amount in each step. The size of the neighborhood is called *window size*. The step size is called *stride*. In practice, for efficiency reasons, convolution is converted into matrix multiplication and addition as well [17], similar to equation 1, except that input and bias vector are matrices: $\mathbf{Y} := \mathbf{W} \cdot \mathbf{X} + \mathbf{B}$.

Dropout and *dropconnect* are types of linear transformations, where multiplication is done elementwise with zero-one random masks [29].

Batch normalization is an adaptive normalization method [29] that shifts outputs $\mathbf{y}$ to amenable ranges. During prediction, batch normalization manifests as a matrix multiplication and addition.

2.1.2 Activation functions. Neural networks use nonlinear transformations of data – *activation functions* – to model nonlinear relationships between input data and output predictions. We identify three common categories:

- *Piecewise linear activation functions.* This category of functions can be represented as a set of n linear functions within specific ranges, each of the type $f_i(y) = a_i y + b_i, y \in [y_i, y_{i+1}]$, where y_i and y_{i+1} are the lower and upper bounds for the range. This category includes the activation functions:

Identity function (linear): $f(\mathrm{y}) = [y_i]$
Rectified Linear Units (ReLU): $f(\mathrm{y}) = [\max(0, y_i)]$
Leaky ReLU: $f(\mathrm{y}) = [\max(0, y_i) + a\min(0, y_i)]$
Maxout (n pieces): $f(\mathrm{y}) = [\max(y_1, \ldots, y_n)]$

- *Smooth activation functions.* A smooth function has continuous derivatives up to some desired order over some domain. Some commonly used smooth activation functions are:

Sigmoid (logistic): $f(\mathrm{y}) = [\frac{1}{1+e^{-y_i}}]$
Hyperbolic tangent (tanh): $f(\mathrm{y}) = [\frac{e^{2y_i}-1}{e^{2y_i}+1}]$
Softplus: $f(\mathrm{y}) = [\log(e^{y_i} + 1)]$

The sigmoid and tanh functions are closely related [29]:

$$tanh(x) = 2 \cdot sigmoid(2x) - 1. \tag{2}$$

They are collectively referred to as *sigmoidal* functions.

- *Softmax*. Softmax is defined as:

$f(\mathrm{y}) = [\frac{e^{y_i}}{\sum_j e^{y_j}}]$

It is usually applied to the last layer to compute a probability distribution in categorical classification. However, in prediction

[1] http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html

phase, usually it is sufficient to use argmax over the outputs of the last layer to predict the most likely outcome.

2.1.3 Pooling operations. Neural networks also commonly use pooling operations that arrange input into several groups and aggregate inputs within each group. Pooling is commonly done by calculating the average or the maximum value among the inputs (*mean* or *max pooling*). Convolution and pooling operations are only used if the input data has spatial structure (e.g., images, sounds).

2.1.4 Commonly used neural network operations. As discussed in Section 2.1.1, all common linear transformations reduce to matrix multiplications and additions in the prediction phase. Therefore it is sufficient for an ONN transformation technique to support making matrix multiplications and additions oblivious.

To get an idea of commonly used activation functions, consider five top performing neural networks[1] in the MNIST [37] and CIFAR-10 [35] datasets. Collectively they support the following activation functions: ReLU [38, 49, 55], leaky ReLU [31, 53], maxout [16, 42] and tanh [18]. In addition, sigmoidal activation functions are commonly used in *language modeling*. Finally, as we saw in Section 2.1.3 common pooling operations are mean and max pooling.

We thus argue that **for an ONN transformation technique to be useful in practice, it should support all of the above commonly used neural network operations**. We describe these in Sections 3 to 5.

Note that although softmax is a popular operation used in the last layer, it can be left out of an ONN [27] (e.g., the input to the softmax layer can be returned to the client) because its application is order-preserving and thus will not change the prediction result.

2.2 Cryptographic preliminaries

2.2.1 Secure two-party computation. Secure two-party computation (2PC) is a type of protocols that allow two parties to jointly compute a function $(f_1(x,y), f_2(x,y)) \leftarrow \mathcal{F}(x,y)$ without learning each other's input. It offers the same security guarantee achieved by a trusted third party TTP running $\mathcal{F}$: both parties submit their inputs (i.e., x and y) to TTP, who computes and returns the corresponding output to each party, so that no information has been leaked except the information that can be inferred from the outputs. Basically, there are three techniques to achieve 2PC: *arithmetic secret sharing* [8], *boolean secret sharing* [28] and Yao's *garbled circuits* [57, 58]. Each technique has its pros and cons, and they can be converted among each other. The ABY framework [20] is a state-of-the-art 2PC library that implements all three techniques.

2.2.2 Homomorphic encryption. A public key encryption scheme is *additively homomorphic* if given two ciphertexts $\hat{x}_1 := E(pk, x_1)$ and $\hat{x}_2 := E(pk, x_2)$, there is a public-key operation $\oplus$ such that $E(pk, x_1 + x_2) \leftarrow \hat{x}_1 \oplus \hat{x}_2$. Examples of such schemes are Paillier's encryption [47], and exponential ElGamal encryption [23]. This kind of encryption schemes is simply referred to as *homomorphic encryption* (HE).

As an inverse of addition, subtraction $\ominus$ is trivially supported by additively homomorphic encryption. Furthermore, adding or multiplying a ciphertext by a constant is efficiently supported: $E(pk, a + x) \leftarrow a \oplus \hat{x}$ and $E(pk, a \cdot x_1) \leftarrow a \otimes \hat{x}_1$.

To do both addition and multiplication between two ciphertexts, fully homomorphic encryption (FHE) or leveled homomorphic encryption (LHE) is needed. However, FHE requires expensive bootstrapping operations and LHE only supports a limited number of homomorphic operations.

2.2.3 Single instruction multiple data (SIMD). The ciphertext of a (homomorphic) encryption scheme is usually much larger than the data being encrypted, and the homomorphic operations on the ciphertexts take longer time than those on the plaintexts. One way to alleviate this issue is to encode several messages into a single plaintext and use the *single instruction multiple data* (SIMD) [52] technique to process these encrypted messages in batch without introducing any extra cost. The LHE library [22] has implemented SIMD based on the Chinese Reminder Theorem (CRT). In this paper, we use $\widetilde{\mathbf{x}}$ to denote the encryption of a vector $[x_1, ..., x_n]$ in batch using the SIMD technique.

The SIMD technique can also be applied to secure two-party computation to reduce the memory footprint of the circuit and improve the circuit evaluation time [11]. In traditional garbled circuits, each wire stores a single input, while in the SIMD version, an input is split across multiple wires so that each wire corresponds to multiple inputs. The ABY framework [20] supports this.

3 PROBLEM STATEMENT

We consider the generic setting for cloud-based prediction services, where a server $\mathcal{S}$ holds a neural network model, and clients $\mathcal{C}$s submit their input to learn corresponding predictions. The model is defined as:

$$\mathbf{z} := (\mathbf{W}_L \cdot f_{L-1}(...f_1(\mathbf{W}_1 \cdot \mathbf{X} + \mathbf{B}_1)...) + \mathbf{b}_L) \quad (3)$$

The problem we tackle is how to design *oblivious neural networks*: after each prediction, $\mathcal{S}$ learns nothing about $\mathbf{X}$, and $\mathcal{C}$ learns nothing about $(\mathbf{W}_1, \mathbf{W}_2, ..., \mathbf{W}_L)$ and $(\mathbf{B}_1, \mathbf{B}_2, ..., \mathbf{b}_L)$ except $\mathbf{z}$. Our security definition follows the standard ideal-world/real-world paradigm: the adversary's view in real-wold is indistinguishable to that in ideal-world.

Adversary model. We assume that either $\mathcal{S}$ or $\mathcal{C}$ can be compromised by an adversary $\mathcal{A}$, but not at the same time. We assume $\mathcal{A}$ to be *semi-honest*, i.e., it directs the corrupted party to follow the protocol specification in real-world, and submits the inputs it received from the environment to TTP in ideal-world. We rely on efficient implementations of primitives (like 2PC in ABY framework [20]) that are secure against semi-honest adversaries.

A compromised $\mathcal{S}$ tries to learn the values in $\mathbf{X}$, and a compromised $\mathcal{C}$ tries to learn the values in $\mathbf{W}$ and $\mathbf{B}$. We do not aim to protect the sizes of $\mathbf{X}$, $\mathbf{W}$, $\mathbf{B}$, and which $f()$ is being used. However, $\mathcal{S}$ can protect such information by adding dummy layers. Note that $\mathcal{C}$s can, in principle, use $\mathcal{S}$'s prediction service as a blackbox oracle to extract an equivalent or near-equivalent model (*model extraction* attacks [54]), or even infer the training set (*model inversion* [25] or *membership inference* attacks [51]). However, in a client-server setting, $\mathcal{S}$ can rate limit prediction requests from a given $\mathcal{C}$, thereby slowing down or bounding this information leakage.

4 MINIONN OVERVIEW

In this section, we explain the basic idea of MiniONN by transforming a toy neural network of the form:

$$\mathbf{z} := \mathbf{W}' \cdot f(\mathbf{W} \cdot \mathbf{x} + \mathbf{b}) + \mathbf{b}' \quad (4)$$

where $\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$, $\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} \\ w_{2,1} & w_{2,2} \end{bmatrix}$, $\mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$, $\mathbf{W}' = \begin{bmatrix} w'_{1,1} & w'_{1,2} \\ w'_{2,1} & w'_{2,2} \end{bmatrix}$ and $\mathbf{b}' = \begin{bmatrix} b'_1 \\ b'_2 \end{bmatrix}$.

The core idea of MiniONN is to have $\mathcal{S}$ and $\mathcal{C}$ *additively share* each of the input *and* output values for every layer of a neural network. That is, at the beginning of every layer, $\mathcal{S}$ and $\mathcal{C}$ will each hold a "share" such that modulo addition of the shares is equal to the input to that layer in the non-oblivious version of that neural network. The output values will be used as inputs for the next layer.

To this end, we have $\mathcal{S}$ and $\mathcal{C}$ first engage in a *precomputation phase* (which is independent of $\mathcal{C}$'s input $\mathbf{x}$), where they jointly generate a set of *dot-product triplets* $\langle u, v, \mathbf{w} \cdot \mathbf{r} \rangle$ for each row of the weight matrices ($\mathbf{W}$ and $\mathbf{W}'$ in this example). Specifically, for each row $\mathbf{w}$, $\mathcal{S}$ and $\mathcal{C}$ run a protocol that securely implements the ideal functionality $\mathcal{F}_{\text{triplet}}$ (in Figure 1) to generate dot-product triplets, such that:

$$\begin{aligned} u_1 + v_1 \pmod N &= w_{1,1}r_1 + w_{1,2}r_2, \\ u_2 + v_2 \pmod N &= w_{2,1}r_1 + w_{2,2}r_2, \\ u'_1 + v'_1 \pmod N &= w'_{1,1}r'_1 + w'_{1,2}r'_2, \\ u'_2 + v'_2 \pmod N &= w'_{2,1}r'_1 + w'_{2,2}r'_2. \end{aligned}$$

Input:

- $\mathcal{S}$: a vector $\mathbf{w} \in \mathbb{Z}_N^n$;
- $\mathcal{C}$: a random vector $\mathbf{r} \in \mathbb{Z}_N^n$.

Output:

- $\mathcal{S}$: a random number $u \in \mathbb{Z}_N$;
- $\mathcal{C}$: $v \in \mathbb{Z}_N$, s.t., $u + v \pmod N = \mathbf{w} \cdot \mathbf{r}$.

Figure 1: Ideal functionality $\mathcal{F}_{\text{triplet}}$: generate a dot-product triplet.

When $\mathcal{C}$ wants to ask $\mathcal{S}$ to compute the predictions for a vector $\mathbf{x} = [x_1, x_2]$, for each x_i, $\mathcal{C}$ chooses a triplet generated in the precomputation phases and uses its r_i value to *blind* x_i.

$$\begin{aligned} x_1^{\mathcal{C}} := r_1, x_1^{\mathcal{S}} &:= x_1 - r_1 \pmod N, \\ x_2^{\mathcal{C}} := r_2, x_2^{\mathcal{S}} &:= x_2 - r_2 \pmod N. \end{aligned}$$

$\mathcal{C}$ then sends $\mathbf{x}^{\mathcal{S}}$ to $\mathcal{S}$, who calculates

$$\begin{aligned} y_1^{\mathcal{S}} &:= w_{1,1}x_1^{\mathcal{S}} + w_{1,2}x_2^{\mathcal{S}} + b_1 + u_1 \pmod N, \\ y_2^{\mathcal{S}} &:= w_{2,1}x_1^{\mathcal{S}} + w_{2,2}x_2^{\mathcal{S}} + b_2 + u_2 \pmod N. \end{aligned}$$

Meanwhile, $\mathcal{C}$ sets:

$$\begin{aligned} y_1^{\mathcal{C}} &:= v_1 \pmod N, \\ y_2^{\mathcal{C}} &:= v_2 \pmod N. \end{aligned}$$

It is clear that

$$\begin{aligned} y_1^{\mathcal{C}} + y_1^{\mathcal{S}} \pmod N &= w_{1,1}x_1 + w_{1,2}x_2 + b_1 \text{ and} \\ y_2^{\mathcal{C}} + y_2^{\mathcal{S}} \pmod N &= w_{2,1}x_1 + w_{2,2}x_2 + b_2. \end{aligned}$$

Therefore, at the end of this interaction, $\mathcal{S}$ and $\mathcal{C}$ additively share the output values $\mathbf{y}$ resulting from the linear transformation in layer 1 without $\mathcal{S}$ learning the input $\mathbf{x}$ and neither party learning $\mathbf{y}$. In Section 5.2 we describe the detailed operations for making linear transformations oblivious.

For the activation/pooling operation $f()$, $\mathcal{S}$ and $\mathcal{C}$ run a protocol that securely implements the ideal functionality in Figure 2, which implicitly reconstructs each $y_i := y_i^{\mathcal{C}} + y_i^{\mathcal{S}} \pmod N$ and returns $x_i^{\mathcal{S}} := f(y_i) - x_i^{\mathcal{C}}$ to $\mathcal{S}$, where $x_i^{\mathcal{C}}$ is $\mathcal{C}$'s component of a previously shared triplet from the precompuation phase, i.e., $x_1^{\mathcal{C}} := r'_1$ and $x_2^{\mathcal{C}} := r'_2$. In Sections 5.3 and 5.4, we show how the ideal functionality in Figure 2 can be concretely realized for commonly used activation functions and pooling operations.

Input:

- $\mathcal{S}$: $y^{\mathcal{S}} \in \mathbb{Z}_N$;
- $\mathcal{C}$: $y^{\mathcal{C}} \in \mathbb{Z}_N$.

Output:

- $\mathcal{S}$: a random number $x^{\mathcal{S}} \in \mathbb{Z}_N$;
- $\mathcal{C}$: $x^{\mathcal{C}} \in \mathbb{Z}_N$ s.t., $x^{\mathcal{C}} + x^{\mathcal{S}} \pmod N = f(y^{\mathcal{S}} + y^{\mathcal{C}} \pmod N))$.

Figure 2: Ideal functionality: oblivious activation/pooling $f()$.

The transformation of the final layer is the same as the first layer. Namely, $\mathcal{S}$ calculates:

$$\begin{aligned} y_1^{\mathcal{S}} &:= w'_{1,1}x_1^{\mathcal{S}} + w'_{1,2}x_2^{\mathcal{S}} + b'_1 + u'_1 \pmod N, \\ y_2^{\mathcal{S}} &:= w'_{2,1}x_1^{\mathcal{S}} + w'_{2,2}x_2^{\mathcal{S}} + b'_2 + u'_2 \pmod N; \end{aligned}$$

and $\mathcal{C}$ sets:

$$\begin{aligned} y_1^{\mathcal{C}} &:= v'_1 \pmod N, \\ y_2^{\mathcal{C}} &:= v'_2 \pmod N. \end{aligned}$$

At the end, $\mathcal{S}$ returns $[y_1^{\mathcal{S}}, y_2^{\mathcal{S}}]$ back to $\mathcal{C}$, who outputs the final predictions:

$$\begin{aligned} z_1 &:= y_1^{\mathcal{C}} + y_1^{\mathcal{S}}, \\ z_2 &:= y_2^{\mathcal{C}} + y_2^{\mathcal{S}}. \end{aligned}$$

Note that MiniONN works in $\mathbb{Z}_N$, while neural networks require floating-point calculations. A simple solution is to scale the floating-point numbers up to integers by multiplying the same constant to all values and drop the fractional parts. A similar technique is used to reduce memory requirements in neural network predictions, at negligible loss of accuracy [41]. We must make sure that the absolute value of any (intermediate) results will not exceed $\lfloor N/2 \rfloor$.

5 MINIONN DESIGN

5.1 Dot-product triplet generation

Recall that we introduce a precomputation phase to generate dot-product triplets, which are similar to the *multiplication triplets* used in secure computations [8]. Multiplication triplets are typically generated in two ways: using homomorphic encryption (HE-based) or using oblivious transfer (OT-based). The former is efficient in terms of communication, whereas the latter is efficient in terms of computation. Both approaches can be optimized for the dot-product generation [43]. In the HE-based approach, dot-products can be calculated directly on ciperhtexts, so that both communication and decryption time can be reduced.

We further improve the HE-based approach using the SIMD batch processing technique. The protocol is described in Figure 3. Using the SIMD technique, $\mathcal{S}$ encrypts the whole vector $\mathbf{w}$ into a single ciphertext of additively homomorphic encryption. $\mathcal{C}$ computes $\widetilde{\mathbf{u}} \leftarrow \mathbf{r} \otimes \widetilde{\mathbf{w}} \ominus \mathbf{v}$, where $\mathbf{r}$ and $\mathbf{v}$ are random vectors generated by $\mathcal{C}$. $\mathcal{S}$ decrypts $\widetilde{\mathbf{u}}$ and outputs the sum of $\mathbf{u}$. Meanwhile, $\mathcal{C}$ outputs the sum of $\mathbf{v}$. Even though $\mathcal{S}$ and $\mathcal{C}$ need to generate new dot-product triplets for each prediction request, $\mathcal{S}$ only needs to transfer $\widetilde{\mathbf{w}}s$ once for all predictions. Furthermore, it can pack multiple $\mathbf{w}$s into a single ciphertext if needed.

Input:
$\mathcal{S}$: $\mathbf{w} \in \mathbb{Z}_N^n$
$\mathcal{C}$: $\mathbf{r} \in \mathbb{Z}_N^n$
Output:
$\mathcal{S}$: a random number $u \in \mathbb{Z}_N$;
$\mathcal{C}$: $v \in \mathbb{Z}_N$, s.t., $u + v \pmod N = \mathbf{w} \cdot \mathbf{r}$.

$\mathcal{S}$:		$\mathcal{C}$:
$\widetilde{\mathbf{w}} \leftarrow E(pk_s, \mathbf{w})$		$\mathbf{v} \xleftarrow{\$} \mathbb{Z}_N^n$
	$\xrightarrow{\widetilde{\mathbf{w}}}$	
		$\widetilde{\mathbf{u}} \leftarrow \mathbf{r} \otimes \widetilde{\mathbf{w}} \ominus \mathbf{v}$
	$\xleftarrow{\widetilde{\mathbf{u}}}$	
$u \leftarrow \sum(D(sk_s, \widetilde{\mathbf{u}}))$		$v \leftarrow \sum(\mathbf{v})$
output u		**output** v

Figure 3: Dot-product triplet generation.

THEOREM 1. *The protocol in Figure 3 securely implements $\mathcal{F}_{\text{triplet}}$ in the presence of semi-honest adversaries, if $E()$ is semantically secure.*

PROOF. Our security proof follows the ideal-world/real-world paradigm: in real-world, parties interact according to the protocol specification, whereas in ideal-world, parties have access to a trusted party TTP that implements $\mathcal{F}_{\text{triplet}}$. The executions in both worlds are coordinated by the environment $\mathcal{E}$, who chooses the inputs to the parties and plays the role of a distinguisher between the real and ideal executions. We aim to show that the adversary's view in real-wold is indistinguishable to that in ideal-world.

Security against a semi-honest server. First, we prove security against a semi-honest server by constructing an ideal-world simulator *Sim* that performs as follows:

(1) receives $\mathbf{w}$ from the environment $\mathcal{E}$; *Sim* sends $\mathbf{w}$ to TTP and gets the result u;
(2) starts running $\mathcal{S}$ on input $\mathbf{w}$, and receives $\widetilde{\mathbf{w}}$;
(3) randomly splits u into a vector $\mathbf{u}'$ s.t., $u = \sum \mathbf{u}'$;
(4) encrypts $\mathbf{u}'$ using $\mathcal{S}$'s public key and returns $\widetilde{\mathbf{u}'}$ to $\mathcal{S}$;
(5) outputs whatever $\mathcal{S}$ outputs.

Next, we show that the view *Sim* simulates for $\mathcal{S}$ is indistinguishable from the view of $\mathcal{S}$ interacting in the real execution. $\mathcal{S}$'s view in the real execution is $\mathbf{u} = \mathbf{w} \cdot \mathbf{r} - \mathbf{v}$ while its view in the ideal execution is $\mathbf{u}' = [r_1', \ldots, r_n']$. So we only need to show that any element $w_i r_i - v_i \pmod N$ in $\mathbf{u}$ is indistinguishable from a random number r_i'. This is clear true since v_i is randomly chosen.

At the end of the simulation, $\mathcal{S}$ outputs $u \leftarrow \sum \mathbf{u}$, which is the same as real execution. Thus, we claim that the output distribution of $\mathcal{E}$ in real-world is computationally indistinguishable from that in ideal-world.

Security against a semi-honest client. Next, we prove security against a semi-honest client by constructing an ideal-world simulator *Sim* that works as follows:

(1) receives $\mathbf{r}$ from $\mathcal{E}$, and sends it to TTP;
(2) starts running $\mathcal{C}$ on input $\mathbf{r}$;
(3) constructs $\widetilde{\mathbf{w}}' \leftarrow E(pk_s', [0, \ldots, 0])$ where pk_s' is randomly generated by *Sim*;
(4) gives $\widetilde{\mathbf{w}}'$ to $\mathcal{C}$;
(5) outputs whatever $\mathcal{C}$ outputs.

$\mathcal{C}$'s view in real execution is $E(pk_s, \mathbf{w})$, which is computationally indistinguishable from its view in ideal execution i.e., $E(pk_s', [0, \ldots, 0])$ due to the semantic security of $E()$. Thus, the output distribution of $\mathcal{E}$ in real-world is computationally indistinguishable from that in ideal-world. □

5.2 Oblivious linear transformations

Recall that when $\mathcal{C}$ wants to request $\mathcal{S}$ to compute predictions for an input $\mathbf{X}$, it blinds each value of $\mathbf{X}$ using a random value r from a dot-product triplet generated earlier: $x^{\mathcal{S}} := x - r \pmod N$. Then, $\mathcal{C}$ sets $\mathbf{X}^{\mathcal{C}} = \mathbf{R}$, and sends $\mathbf{X}^{\mathcal{S}}$ to $\mathcal{S}$. The security of the dot-product generation protocol guarantees that $\mathcal{S}$ knows nothing about the r values. Consequently, $\mathcal{S}$ cannot get any information about $\mathbf{X}$ from $\mathbf{X}^{\mathcal{S}}$ if all rs are randomly chosen by $\mathcal{C}$ from $\mathbb{Z}_N$.

Upon receiving $\mathbf{X}^{\mathcal{S}}$, $\mathcal{S}$ will input it to the first layer which is typically a linear transformation layer. As we discussed in Section 2.1, all linear transformations can be turned into matrix multiplications/additions: $\mathbf{Y} = \mathbf{W} \cdot \mathbf{X} + \mathbf{B}$. Figure 4 shows the oblivious linear transformation protocol. For each row of $\mathbf{W}$ and each column of $\mathbf{X}^{\mathcal{C}}$, $\mathcal{S}$ and $\mathcal{C}$ jointly generate a dot-product triplet: $u + v \pmod N = \mathbf{w} \cdot \mathbf{x}^{\mathcal{C}}$. Since $\mathbf{X}^{\mathcal{C}}$ is independent of $\mathbf{X}$, they can generate such triplets in a precomputation phase. Next, $\mathcal{S}$ calculates $\mathbf{Y}^{\mathcal{S}} := \mathbf{W} \cdot \mathbf{X}^{\mathcal{S}} + \mathbf{B} + \mathbf{U}$, and meanwhile $\mathcal{C}$ sets $\mathbf{Y}^{\mathcal{C}} := \mathbf{V}$. Consequently, each element of $\mathbf{Y}^{\mathcal{S}}$ and $\mathbf{Y}^{\mathcal{C}}$ satisfy:

$$\begin{aligned} y^{\mathcal{S}} + y^{\mathcal{C}} &= \mathbf{w} \cdot \mathbf{x}^{\mathcal{S}} + b + u + v \\ &= w_1(x_1 - x_1^{\mathcal{C}}) +, \ldots, + w_l(x_l - x_l^{\mathcal{C}}) + b + u + v \\ &= (w_1 x_1 +, \ldots, + w_l x_l + b) - (w_1 x_1^{\mathcal{C}} +, \ldots, + w_l x_l^{\mathcal{C}}) + u + v \\ &= y \end{aligned}$$

Due to the fact that $\langle \mathbf{U}, \mathbf{V} \rangle$ are securely generated by $\mathcal{F}_{\text{triplet}}$, the outputs of this layer (which are the inputs to the next layer) are also randomly shared between $\mathcal{S}$ and $\mathcal{C}$, i.e., $\mathbf{Y}^{\mathcal{C}} = \mathbf{V}$ and $\mathbf{Y}^{\mathcal{S}} = \mathbf{Y} - \mathbf{V}$ can be used as inputs for the next layer directly.

It is clear that the view of both $\mathcal{S}$ and $\mathcal{C}$ are identical to their views under the dot-product triplet generation protocol. Therefore, the oblivious linear transformation protocol is secure if $\mathcal{F}_{\text{triplet}}$ is securely implemented.

A linear transformation layer can also follow an activation layer or a pooling layer. So, we need to design the oblivious activation/pooling operations in a way that their outputs can be the

Input:
$\mathcal{S}$: $\mathbf{W} \in \mathbb{Z}_N^{m\times l}$, $\mathbf{X}^{\mathcal{S}} \in \mathbb{Z}_N^{l\times n}$, $\mathbf{B} \in \mathbb{Z}_N^{m\times n}$
$\mathcal{C}$: $\mathbf{X}^{\mathcal{C}} \in \mathbb{Z}_N^{l\times n}$
Output:
$\mathcal{S}$: A random matrix $\mathbf{Y}^{\mathcal{S}}$
$\mathcal{C}$: $\mathbf{Y}^{\mathcal{C}}$ s.t., $\mathbf{Y}^{\mathcal{C}} + \mathbf{Y}^{\mathcal{S}} = \mathbf{W} \cdot (\mathbf{X}^{\mathcal{C}} + \mathbf{X}^{\mathcal{S}}) + \mathbf{B}$

$\mathcal{S}$: $\mathcal{C}$:

precomputation

for $i = 1$ **to** m
for $j = 1$ **to** n
$(u_{i,j}, v_{i,j}) \leftarrow \mathcal{F}_{\text{triplet}}(\mathbf{w}_i, \mathbf{x}_j^{\mathcal{C}})$
end
end

$\mathbf{Y}^{\mathcal{S}} := \mathbf{W} \cdot \mathbf{X}^{\mathcal{S}} + \mathbf{B} + \mathbf{U}$ $\quad$ $\mathbf{Y}^{\mathcal{C}} := \mathbf{V}$
output $\mathbf{Y}^{\mathcal{S}}$ $\quad$ **output** $\mathbf{Y}^{\mathcal{C}}$

Figure 4: Oblivious linear transformation.

inputs to linear transformations: $\mathbf{X}^{\mathcal{S}}$ and $\mathbf{X}^{\mathcal{C}}$ s.t. $\mathbf{X}^{\mathcal{S}} + \mathbf{X}^{\mathcal{C}} = \mathbf{X}$ and $\mathbf{X}^{\mathcal{C}}$ has been used to generate the dot-product triplets for the next layer. See the following sections.

5.3 Oblivious activation functions

In this section, we introduce the oblivious activation function which receives $y^{\mathcal{C}}$ from $\mathcal{C}$ and $y^{\mathcal{S}}$ from $\mathcal{S}$, and outputs $x^{\mathcal{C}}$ to $\mathcal{C}$ and $x^{\mathcal{S}} := f(y^{\mathcal{S}} + y^{\mathcal{C}}) - x^{\mathcal{C}}$ to $\mathcal{S}$, where $x^{\mathcal{C}}$ is a random number generated by $\mathcal{C}$. Note that if the next layer is a linear transformation layer, $x^{\mathcal{C}}$ should be the random value that has been used by $\mathcal{C}$ to generate a dot-product triplet in the precompuation phase. On the other hand, if the next layer is a pooling layer, $x^{\mathcal{C}}$ can be generated on demand.

5.3.1 Oblivious piecewise linear activation functions. Piecewise linear activation functions are widely used in image classifications due to their outstanding performance in training phase as demonstrated by Krizhevsky et al. [36]. We take ReLU as an example to illustrate how to transform piecewise linear functions into their oblivious forms. Recall that ReLU is $f(y) = max(0, y)$, where y is additively shared between $\mathcal{S}$ and $\mathcal{C}$. An oblivious ReLU protocol will reconstruct y and return $max(0, y) - x^{\mathcal{C}}$ to $\mathcal{S}$. This is equivalent to the ideal functionality $\mathcal{F}_{\text{ReLU}}$ in Figure 5. Actually, we compare y with $\frac{N}{2}$: $y > \frac{N}{2}$ implies y is negative (recall that absolute values of all intermediate results will not exceed $\lfloor N/2 \rfloor$).

$\mathcal{F}_{\text{ReLU}}$ can be trivially implemented by a 2PC protocol. Specifically, we use a garbled circuit to reconstruct y and calculate $b := compare(y, 0)$ to determine whether $y \geq 0$ or not. If $y \geq 0$, it returns y, otherwise, it returns 0. This is achieved by multiplying y with b. The only operations we need for oblivious ReLU are $+, -, \cdot$ and *compare*, all of which are supported by the 2PC library [20] we used. So both implementation and security argument are straightforward.

Oblivious leaky ReLU can be constructed in the same way as oblivious ReLU, except that $\mathcal{S}$ gets:

Input:
- $\mathcal{S}$: $y^{\mathcal{S}} \in \mathbb{Z}_N$;
- $\mathcal{C}$: $y^{\mathcal{C}}, r \in \mathbb{Z}_N$.

Output:
- $\mathcal{S}$: $x^{\mathcal{S}} := compare(y, 0) \cdot y - r \pmod N$ where $y = y^{\mathcal{S}} + y^{\mathcal{C}} \pmod N$;
- $\mathcal{C}$: $x^{\mathcal{C}} := r$.

Figure 5: The ideal functionality $\mathcal{F}_{\text{ReLU}}$.

$$x^{\mathcal{S}} := compare(y, 0) \cdot a \cdot y + (1 - compare(y, 0)) \cdot y - r \pmod N.$$

5.3.2 Oblivious smooth activation functions. Unlike piecewise linear functions, it is non-trivial to make smooth functions oblivious. For example, in the sigmoid function $f(y) = \frac{1}{1+e^{-y}}$, both e^y and division are expensive to be computed by 2PC protocols [48]. Furthermore, it is difficult to keep track of the floating point value of e^y, especially when y is blinded. It is well-known that such functions can be approximated locally by high-degree polynomials, but oblivious protocols can only handle low-degree approximation polynomials efficiently. To this end, we adapt an approximation method that can be efficiently computed by an oblivious protocol and incurs negligible accuracy loss.

Approximation of smooth functions. A smooth function $f()$ can be approximated by a set of piecewise continuous polynomials, i.e., *splines* [21]. The idea is to split $f()$ into several intervals, in each of which, a polynomial is used to to approximate $f()$. The polynomials are chosen such that the overall goodness of fit is maximized. The approximation method is detailed in the following steps:

(1) Set the approximation range $[\alpha_1, \alpha_n]$, select n equally spaced samples (including α_1 and α_n). The resulting sample set is $\{\alpha_1, ..., \alpha_n\}$
(2) For each α_i, calculate $\beta_i := f(\alpha_i)$.
(3) Find m switchover positions (i.e., knots) for polynomials expressions:
   (a) fit an initial approximation $\bar{f}$ of order d for the dataset $\{\alpha_i, \beta_i\}$ using polynomial regression (without knots);
   (b) select a new knot $\dot{\alpha}_i \in \{\alpha_1, \ldots, \alpha_n\}$ and fit two new polynomial expressions on each side of the knot (the knot is chosen such that the overall goodness of fit is maximized);
   (c) repeat (b) until the number of knots equals m.

   The set of knots is now $\{\dot{\alpha}_1, ..., \dot{\alpha}_m\}$. Note that $\dot{\alpha}_1 = \alpha_1$ and $\dot{\alpha}_m = \alpha_n$.
(4) Fit a smoothing spline ([21], Chapter 5) of the same order using the knots $\{\dot{\alpha}_i\}$ on the dataset $\{\alpha_i, \beta_i\}$ and extract the polynomial expression $P_i(\alpha)$ in the each interval $[\dot{\alpha}_i, \dot{\alpha}_{i+1}], i \in \{1, m-1\}$.[2]
(5) Set boundary polynomials $P_0()$ (for $\alpha < \dot{\alpha}_1$) and $P_m()$ (for $\alpha > \dot{\alpha}_m$), which are chosen specifically for $f()$ to closely approximate the behaviour beyond the ranges $[\alpha_1, \alpha_n]$.

[2] We use the functions in the library scipy.interpolate.UnivariateSpline and numpy.polyfit [33]

Thus, we split $f()$ into $m + 1$ intervals, and each has a separate polynomial expression.[3]

(6) The final approximation is:

$$\bar{f}(\alpha) = \begin{cases} P_0(\alpha) & \text{if } \alpha < \dot{\alpha}_1 \\ P_1(\alpha) & \text{if } \dot{\alpha}_1 \leq \alpha < \dot{\alpha}_2 \\ \ldots & \\ P_{m-1}(\alpha) & \text{if } \dot{\alpha}_{m-1} \leq \alpha < \dot{\alpha}_m \\ P_m(\alpha) & \text{if } \alpha \geq \dot{\alpha}_m, \end{cases} \tag{5}$$

Note that any univariate monotonic functions can be fitted by above procedure.

Oblivious approximated sigmoid. We take sigmoid as an example to explain how to transform smooth activation functions into their oblivious forms. We set the polynomial degree d as 1, since linear functions (as opposed to higher-degree polynomials) are faster and less memory-consuming to be computed by 2PC. The approximated sigmoid function is as follows:

$$\bar{f}(y) = \begin{cases} 0 & \text{if } y < y_1 \\ a_1 y + b_1 & \text{if } y_1 \leq y < y_2 \\ \ldots & \\ a_{m-1} y + b_{m-1} & \text{if } y_{m-1} \leq y < y_m \\ 1 & \text{if } y \geq y_m, \end{cases} \tag{6}$$

We will show (in Section 6.2) that it approximates sigmoid with negligible accuracy loss.

The approximated sigmoid function (Equation 6) is in fact a piecewise linear function. So it can be transformed in the same way as we explained in Section 5.3.1. The ideal functionality for the approximated sigmoid $\mathcal{F}_{\text{sigmoid}}$ is shown in Figure 6. Correctness of this functionality follows the fact that, for $y_i \leq y < y_{i+1}$:

$$x = ((a_i y + b_i) - (a_{i+1} y + b_{i+1})) + ((a_i y + b_i) - (a_{i+1} y + b_{i+1})) \\ +\ldots + ((a_{m-1} y + b_{m-1}) - 1) + 1$$

Input:

- $\mathcal{S}$: $y^{\mathcal{S}} \in \mathbb{Z}_N$;
- $\mathcal{C}$: $y^{\mathcal{C}}, r \in \mathbb{Z}_N$.

Output:

- $\mathcal{S}$: $x^{\mathcal{S}} := compare(y_1, y) \cdot (0 - (a_1 y + b_1))$
  $+compare(y_2, y) \cdot ((a_1 y + b_1) - (a_2 y + b_2))$
  $\ldots$
  $+compare(y_{m-1}, y) \cdot ((a_{m-1} y + b_{m-1}) - 1) + 1$
  $-r \pmod N$, where $y = y^{\mathcal{S}} + y^{\mathcal{C}} \pmod N$;
- $\mathcal{C}$: $x^{\mathcal{C}} := r$.

Figure 6: The ideal functionality $\mathcal{F}_{\text{sigmoid}}$.

Even though it is more complex than $\mathcal{F}_{\text{ReLU}}$, it can still be realized easily using the basic functionalities provided by 2PC.

In summary, we support all activation functions that are both:

(1) monotonic in ranges $(\infty, 0)$ and $(0, \infty)$;[4] and
(2) either piecewise-polynomial or approximable.

Most commonly used activation functions belong to this class, except softmax that violates the second condition. But softmax can be replaced by argmax (Section 2.1.2) or left out (Section 2.1).

5.4 Oblivious pooling operations

The pooling layer arranges the inputs into several groups and take the max or mean of the elements in each group. For mean pooling, we just have $\mathcal{S}$ and $\mathcal{C}$ calculate the sum of their respective shares and keep track of the divisor. For max pooling, we use garbled circuits to realize the ideal functionality $\mathcal{F}_{\max}$ in Figure 7, which reconstructs each y_i and returns the largest one masked by a random number. The *max* function can be easily achieved by the *compare* function.

Inputs:

- $\mathcal{S}$: $\{y_1^{\mathcal{S}}, \ldots, y_n^{\mathcal{S}}\}$;
- $\mathcal{C}$: $\{y_1^{\mathcal{C}}, \ldots, y_n^{\mathcal{C}}\}$, r.

Outputs:

- $\mathcal{S}$: $x^{\mathcal{S}} := max(y_1, \ldots, y_n) - r \pmod N$ where $y_1 = y_1^{\mathcal{S}} + y_1^{\mathcal{C}} \pmod N$ ... $y_n = y_n^{\mathcal{S}} + y_n^{\mathcal{C}} \pmod N$;
- $\mathcal{C}$: $x^{\mathcal{C}} := r$.

Figure 7: The ideal functionality $\mathcal{F}_{\max}$.

Note that the oblivious *maxout* activation can be trivially realized by the ideal functionality $\mathcal{F}_{\max}$.

5.5 Remarks

5.5.1 Oblivious square function. The square function (i.e., $f(y) = y^2$) is also used as an activation function in [27, 43], because it is easier to be transformed into an oblivious form. We implement an oblivious square function by realizing the ideal functionality in Figure 8 using arithmetic secret sharing.

Input:

- $\mathcal{S}$: $y^{\mathcal{S}} \in \mathbb{Z}_N$;
- $\mathcal{C}$: $y^{\mathcal{C}}, r \in \mathbb{Z}_N$.

Output:

- $\mathcal{S}$: $x^{\mathcal{S}} := y^2 - r \pmod N$ where $y = y^{\mathcal{S}} + y^{\mathcal{C}} \pmod N$;
- $\mathcal{C}$: $x^{\mathcal{C}} := r$.

Figure 8: The ideal functionality $\mathcal{F}_{\text{Square}}$.

5.5.2 Dealing with large numbers. Recall that we must make sure that the absolute value of any (intermediate) results will not exceed $\lfloor N/2 \rfloor$. However, the data range grows exponentially with the number of multiplications, and it grows even faster when the floating point numbers are scaled to integers. Furthermore, the SIMD

[3]We apply post-processing to the polynomials to ensure they are within upper and lower bounds of the function $f()$, and to ensure that the approximate function $\bar{f}$ is monotonic (if $f()$ is).

[4]This condition guarantees that our scaling technique (i.e., scale the floating-point numbers up to integers by multiplying the same constant to all values and drop the fractional parts) does not change the ranking order and thus does not impact prediction accuracy. Recall that the model finally outputs a set of probabilities, one for each class. The class with maximal probability is the prediction.

technique will shrink the plaintext space so that it cannot encrypt large numbers. As a result, only a limited number of multiplications can be supported.

To this end, CryptoNets uses Chinese Remainder Theorem (CRT) to split large numbers into multiple small parts, work on each part individually, and combines the results in the end [27]. This method allows encryptions of exponentially large numbers in linear time and space, but the overhead grows linearly with the number of split parts. On the other hand, SecureML has both parties truncate their individual shares independently [43]. This method may incur a small error in each intermediate result, which may affect the final prediction accuracy.

We implement the ideal functionality in Figure 9 using garbled circuit to securely scale down the data range without affecting accuracy. It reconstructs y and shift it left by L bits, where L is a constant known to both $\mathcal{S}$ and $\mathcal{C}$. This is equivalent to $x^{\mathcal{S}} := \left\lfloor \frac{y}{2^L} \right\rfloor - r \pmod N$. They can run this protocol after each layer, or whenever needed.

Input:

- $\mathcal{S}$: $y^{\mathcal{S}} \in \mathbb{Z}_N$;
- $\mathcal{C}$: $y^{\mathcal{C}}, r \in \mathbb{Z}_N$.

Output:

- $\mathcal{S}$: $x^{\mathcal{S}} := \textit{leftshift}\,(y, L) - r \pmod N$ where $y = y^{\mathcal{S}} + y^{\mathcal{C}} \pmod N$;
- $\mathcal{C}$: $x^{\mathcal{C}} := r$.

Figure 9: The ideal functionality $\mathcal{F}_{\text{trunc}}$.

5.5.3 Security. We have proved the security for the dot-product triplet generation, and other operations are directly implemented using 2PC protocols. So the security guarantees of all operations are straightforward. Furthermore, the output of each operation is randomly shared between $\mathcal{S}$ and $\mathcal{C}$. As stated in Lemma 2 of [**?**], a protocol that ends with secure re-sharing of output is universally composable. So, after composition, the entire ONN is secure.

6 PERFORMANCE EVALUATION

We implemented MiniONN in C++ using Boost[5] for networking. We used the ABY [20] library for secure two-party computation with 128-bit security parameter and SIMD circuits. We used YASHE [12] for additively homomorphic encryption, a SIMD version of which is supported by the SEAL library [22]. The YASHE encryption scheme maps plaintext messages from the ring $\mathbb{Z}_N[x]/(x^n + 1)$ to the ring $\mathbb{Z}_q[x]/(x^n + 1)$. The ciphertext modulus q determines the security level. The SEAL library automatically chooses a secure q given the polynomial degree n (i.e., SIMD batch size). Choice of n is a tradeoff between parallelism and efficiency of a single encryption. We first set n =4 096 so that we can encrypt 4 096 elements together in a reasonable encryption time and ciphertext size. Then we chose the largest possible plaintext modulus: N =101 285 036 033, which is large enough for the needed precision since we securely scale down the value when it becomes large as we discussed in Section 5.5.2.

To evaluate its performance, we ran the server-side program on a remote computer (Intel Core i5 CPU with 4 3.30 GHz cores and 16 GB memory) and the client-side program on a local desktop (Intel Core i5 CPU machine with 4 3.20 GHz cores and 8 GB memory). We used the *Clocks* module in C++ for time measurement and used *TCPdump* for bandwidth measurement. We measured response latency (including the network delay) and message sizes during the whole procedure, i.e., from the time $\mathcal{C}$ begins to generate its request to the time it obtains the final predictions. Each experiment was repeated 5 times and we calculated the mean and standard deviation. The standard deviations in all reported results are less than 3%.

6.1 Comparisons with previous work

The MNIST dataset [37] consists of 70 000 black-white hand-written digit images (of size $1 \times 28 \times 28$: width and height are 28 pixels) in 10 classes. There are 60 000 training images and 10 000 test images. Since previous work use MNIST to evaluate their techniques, we use it to provide a direct comparison with prior work..

Neural network in SecureML [43]. We reproduced the model (Figure 10) presented in SecureML [43]. It uses multi-layer perceptron (MLP) model with square as the activation function and achieves an accuracy of 93.1% in the MNIST dataset. We improve the accuracy of this model to 97.6% by using the *Limited-memory BFGS* [39] optimization algorithm and batch normalization during training. We transformed this model with MiniONN and compared the results with those reported in [27].

(1) *Fully Connected:* input image 28×28, connects the incoming 784 nodes to the outgoing 128 nodes: $\mathbb{R}^{128\times1} \leftarrow \mathbb{R}^{128\times784} \cdot \mathbb{R}^{784\times1}$.
(2) *Square Activation:* squares the value of each input.
(3) *Fully Connected:* connects the incoming 128 nodes to the outgoing 128 nodes: $\mathbb{R}^{128\times1} \leftarrow \mathbb{R}^{128\times128} \cdot \mathbb{R}^{128\times1}$.
(4) *Square Activation:* squares the value of each input.
(5) *Fully Connected:* fully connects the incoming 128 nodes to the outgoing 10 nodes: $\mathbb{R}^{10\times1} \leftarrow \mathbb{R}^{10\times128} \cdot \mathbb{R}^{128\times1}$.

Figure 10: The neural network presented in SecureML [43].

The results (Table 2) show that MiniONN achieves comparable online performance and significantly better offline performance. We take the first layer as an example to explain why the SIMD batch processing technique improves the performance of offline phase. The first layer connects 784 incoming nodes to 128 outgoing nodes, which leads to a matrix multiplication: $\mathbb{R}^{128\times1} \leftarrow \mathbb{R}^{128\times784} \cdot \mathbb{R}^{784\times1}$. In SecureML [43], $\mathcal{C}$ encrypts each of the 784 elements separately and sends them to $\mathcal{S}$, which leads to 784 encryptions and ciphertext transfers. $\mathcal{S}$ applies each row of the matrix to the ciphertexts to calculate an encrypted dot-product, which leads to $784 \times 128 = 100\,352$ homomorphic multiplications. Then $\mathcal{S}$ returns the resulting 128 ciphetexts to $\mathcal{C}$, who decrypts them, which leads to another 128 ciphertext transfers and 128 decryptions. On the other hand, we duplicate the 784 elements into 128 copies, and encrypt them into 25 ciphertexts, since each ciphertext can pack 4096 elements. $\mathcal{S}$ encodes the matrix into 25 batches and multiplies them to the ciphertexts, which only leads to 25 homomorphic multiplications. Table 3 summarizes this comparison.

[5] http://www.boost.org

Square/MLP/MNIST (Figure 10)	Latency (s)		Message Sizes (MB)		Accuracy %
	offline	online	offline	online	
by SecureML [43]	4.7	0.18	-	-	93.1
by MiniONN	0.9	0.14	3.8	12	97.6

Table 2: Comparison: MiniONN vs. SecureML [43].

	SecureML [43]	MiniONN
# homomorphic encryptions	784	25
# homomorphic multiplications	100 352	25
# ciphertext transfers	912	50
# homomorphic decryptions	128	25

Table 3: Comparison: MiniONN vs. SecureML [43], dot-product triplet generations.

Neural network in CryptoNets [27]. We reproduced the model (Figure 11) presented in CryptoNets [27]. It is a CNN model with square as the activation function as well, and uses mean pooling instead of max pooling. Due to the convolution operation, it achieves a higher accuracy of 98.95% in the MNIST dataset. We transformed this model with MiniONN and compared its performance with the results reported in CryptoNets [27]. Table 4 shows that MiniONN achieves 230-fold reduction in latency and 8-fold reduction in message sizes, without degradation in accuracy. CryptoNets uses the SIMD technique to batch different requests to achieve a throughput of 51 739 predictions per hour, but these requests must be from the *same* client. In scenarios where the same client sends a very large number of prediction requests and can tolerate response latency in the order of minutes, CryptoNets can achieve 6-fold throughput than MiniONN. In scenarios where each client sends only a small number of requests but needs quick responses, MiniONN decisively outperforms CryptoNets.

(1) *Convolution:* input image 28×28, window size 5×5, stride (2, 2), number of output channels 5. It can be converted to matrix multiplication [17]: $\mathbb{R}^{5\times169} \leftarrow \mathbb{R}^{5\times25} \cdot \mathbb{R}^{25\times169}$.
(2) *Square Activation:* squares the value of each input.
(3) *Pool:* combination of mean pooling and linear transformation: $\mathbb{R}^{100\times1} \leftarrow \mathbb{R}^{100\times845} \cdot \mathbb{R}^{845\times1}$.
(4) *Square Activation:* squares the value of each input.
(5) *Fully Connected:* fully connects the incoming 100 nodes to the outgoing 10 nodes: $\mathbb{R}^{10\times1} \leftarrow \mathbb{R}^{10\times100} \cdot \mathbb{R}^{100\times1}$.

Figure 11: The neural network presented in CryptoNets [27].

Square/CNN/MNIST (Figure 11)	Latency (s)		Message Sizes (MB)		Accuracy %
	offline	online	offline	online	
by CryptoNets [27]	0	297.5	0	372.2	98.95
by MiniONN	0.88	0.4	3.6	44	98.95

Table 4: Comparison: MiniONN vs. CryptoNets [27].

6.2 Evaluations with realistic models

As we stated in Section 2, a useful ONN transformation technique must support commonly used neural network operations. Both CryptoNets and SecureML [43] fall short on this count. In this section we discuss performance evaluations of realistic models that are built with popular neural network operations using several different standard datasets.

Handwriting recognition: MNIST. We trained and implemented another neural network (Figure 12) using the MNIST dataset, but using ReLU as the activation function. The use of ReLU with a more complex neural network increases the accuracy of the model in MNIST to 99.31%, which is close to the state-of-the-art accuracy in the MNIST dataset (99.79%)[6].

(1) *Convolution:* input image 28×28, window size 5×5, stride (1, 1), number of output channels of 16: $\mathbb{R}^{16\times576} \leftarrow \mathbb{R}^{16\times25} \cdot \mathbb{R}^{25\times576}$.
(2) *ReLU Activation:* calculates ReLU for each input.
(3) *Max Pooling:* window size $1 \times 2 \times 2$ and outputs $\mathbb{R}^{16\times12\times12}$.
(4) *Convolution:* window size 5×5, stride (1, 1), number of output channels 16: $\mathbb{R}^{16\times64} \leftarrow \mathbb{R}^{16\times400} \cdot \mathbb{R}^{400\times64}$.
(5) *ReLU Activation:* calculates ReLU for each input.
(6) *Max Pooling:* window size $1 \times 2 \times 2$ and outputs $\mathbb{R}^{16\times4\times4}$.
(7) *Fully Connected:* fully connects the incoming 256 nodes to the outgoing 100 nodes: $\mathbb{R}^{100\times1} \leftarrow \mathbb{R}^{100\times256} \cdot \mathbb{R}^{256\times1}$.
(8) *ReLU Activation:* calculates ReLU for each input
(9) *Fully Connected:* fully connects the incoming 100 nodes to the outgoing 10 nodes: $\mathbb{R}^{10\times1} \leftarrow \mathbb{R}^{10\times100} \cdot \mathbb{R}^{100\times1}$.

Figure 12: The neural network trained from the MNIST dataset.

Image classification: CIFAR-10. CIFAR-10 [35] is a standard dataset consisting of RGB images (of size $3 \times 32 \times 32$, 3 color channels, width and height are 32) of everyday objects in 10 classes (e.g., automobile, bird etc.). The training set has 50 000 images while the test set has 10 000 images. The neural network is detailed in Figure 13. It achieves 81.61% prediction accuracy.

(1) *Convolution:* input image $3 \times 32 \times 32$, window size 3×3, stride (1, 1), pad (1, 1), number of output channels 64: $\mathbb{R}^{64\times1024} \leftarrow \mathbb{R}^{64\times27} \cdot \mathbb{R}^{27\times1024}$.
(2) *ReLU Activation:* calculates ReLU for each input.
(3) *Convolution:* window size 3×3, stride (1, 1), pad (1, 1), number of output channels 64: $\mathbb{R}^{64\times1024} \leftarrow \mathbb{R}^{64\times576} \cdot \mathbb{R}^{576\times1024}$.
(4) *ReLU Activation:* calculates ReLU for each input.
(5) *Mean Pooling:* window size $1 \times 2 \times 2$, outputs $\mathbb{R}^{64\times16\times16}$.
(6) *Convolution:* window size 3×3, stride (1, 1), pad (1, 1), number of output channels 64: $\mathbb{R}^{64\times256} \leftarrow \mathbb{R}^{64\times576} \cdot \mathbb{R}^{576\times256}$.
(7) *ReLU Activation:* calculates ReLU for each input.
(8) *Convolution:* window size 3×3, stride (1, 1), pad (1, 1), number of output channels 64: $\mathbb{R}^{64\times256} \leftarrow \mathbb{R}^{64\times576} \cdot \mathbb{R}^{576\times256}$.
(9) *ReLU Activation:* calculates ReLU for each input.
(10) *Mean Pooling:* window size $1 \times 2 \times 2$, outputs $\mathbb{R}^{64\times16\times16}$.
(11) *Convolution:* window size 3×3, stride (1, 1), pad (1, 1), number of output channels 64: $\mathbb{R}^{64\times64} \leftarrow \mathbb{R}^{64\times576} \cdot \mathbb{R}^{576\times64}$.
(12) *ReLU Activation:* calculates ReLU for each input.
(13) *Convolution:* window size 1×1, stride (1, 1), number of output channels of 64: $\mathbb{R}^{64\times64} \leftarrow \mathbb{R}^{64\times64} \cdot \mathbb{R}^{64\times64}$.
(14) *ReLU Activation:* calculates ReLU for each input.
(15) *Convolution:* window size 1×1, stride (1, 1), number of output channels of 16: $\mathbb{R}^{16\times64} \leftarrow \mathbb{R}^{16\times64} \cdot \mathbb{R}^{64\times64}$.
(16) *ReLU Activation:* calculates ReLU for each input.
(17) *Fully Connected Layer:* fully connects the incoming 1024 nodes to the outgoing 10 nodes: $\mathbb{R}^{10\times1} \leftarrow \mathbb{R}^{10\times1024} \cdot \mathbb{R}^{1024\times1}$.

Figure 13: The neural network trained from the CIFAR-10 dataset.

Language modeling: PTB. Penn Treebank (PTB) is a standard dataset [40] for *language modeling*, i.e., predicting *likely* next words given the

[6]http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html (last accessed May 9, 2017)

previous words ([44], Chapter 27). We used a preprocessed version of this dataset[7], which consists of 929 000 training words, 73 000 validation words, and 82 000 test words.

Long Short Term Memory (LSTM) is a neural network architecture that is commonly used for language modeling [32]. Sigmoidal activation functions are typically used in such networks. We reproduced and transformed a recent LSTM model [59] following the tutorial[8] in Tensorflow [1]. To the extent of our knowledge, this is the first time language modeling is performed using oblivious models, which paves the way to oblivious neural machine translation. The model is described in Figure 14.

(1) *Fully Connected:* input one-hot vector word 10000×1, fully connects the input nodes to the outgoing 200 nodes: $\mathbb{R}^{200\times1} \leftarrow \mathbb{R}^{200\times10000} \cdot \mathbb{R}^{10000\times1}$.
(2) *LSTM:* pad the incoming 200 nodes with another 200 nodes: $\mathbb{R}^{400\times1} \leftarrow \mathbb{R}^{200\times1} \mid \mathbb{R}^{200\times1}$, and then process them as follows:
   (a) $\mathbb{R}^{800\times1} \leftarrow \mathbb{R}^{800\times400} \cdot \mathbb{R}^{400\times1}$
   (b) $\mathbb{R}^{200\times1} \mid \mathbb{R}^{200\times1} \mid \mathbb{R}^{200\times1} \mid \mathbb{R}^{200\times1} \leftarrow \mathbb{R}^{800\times1}$
   (c) $\mathbb{R}^{200} \leftarrow \mathbb{R}^{200} \circ \mathit{sigmoid}(\mathbb{R}^{200}) + \mathit{sigmoid}(\mathbb{R}^{200}) \circ \mathit{tanh}(\mathbb{R}^{200})$
   (d) $\mathbb{R}^{200} \leftarrow \mathit{sigmoid}(\mathbb{R}^{200}) \circ \mathit{tanh}(\mathbb{R}^{200})$
(3) *LSTM:* pad the incoming 200 nodes with another 200 nodes: $\mathbb{R}^{400\times1} \leftarrow \mathbb{R}^{200\times1} \mid \mathbb{R}^{200\times1}$, and then process them as follows:
   (a) $\mathbb{R}^{800\times1} \leftarrow \mathbb{R}^{800\times400} \cdot \mathbb{R}^{400\times1}$
   (b) $\mathbb{R}^{200\times1} \mid \mathbb{R}^{200\times1} \mid \mathbb{R}^{200\times1} \mid \mathbb{R}^{200\times1} \leftarrow \mathbb{R}^{800\times1}$
   (c) $\mathbb{R}^{200} \leftarrow \mathbb{R}^{200} \circ \mathit{sigmoid}(\mathbb{R}^{200}) + \mathit{sigmoid}(\mathbb{R}^{200}) \circ \mathit{tanh}(\mathbb{R}^{200})$
   (d) $\mathbb{R}^{200} \leftarrow \mathit{sigmoid}(\mathbb{R}^{200}) \circ \mathit{tanh}(\mathbb{R}^{200})$
(4) *Fully Connected:* fully connects the incoming 200 nodes to the outgoing 10000 nodes: $\mathbb{R}^{10000\times1} \leftarrow \mathbb{R}^{10000\times200} \cdot \mathbb{R}^{200\times1}$.

Figure 14: The neural network trained from the PTB dataset.

We used the real sigmoid activation functions for training, but replaced them with their corresponding approximations (Section 5.3.2) for predictions. In our sigmoid approximation, we set the ranges as $[\alpha_0, \alpha_n] = [-30, 30]$ and set the polynomials beyond the ranges as 0 and 1, i.e., $\bar{f}(y < -30) := 0$ and $\bar{f}(y > 30) := 1$ as in Equation 6.[9] Unlike aforementioned image datasets, prediction quality here is measured by a loss function called *cross-entropy loss* [44]. Figure 15 shows that the cross-entropy loss achieved by our approximation method (with more than 12 pieces) is close to the original result (4.76 vs. 4.74). We also test the new activation function that is proposed in SecureML [43] as an alternative to the sigmoid function. In this model, it causes the cross-entropy loss to diverge to infinity. The optimal number of linear pieces differs on the model structure, e.g., 14 pieces achieved optimal results on the larger models in [59].

Summary of results. Table 5 summarizes the results of the last three neural networks after being transformed by MiniONN. The performance of the ONNs in MNIST and PTB is reasonable, whereas the ONN in CIFAR-10 is too expensive. This is due to the fact that the model in CIFAR-10 (Figure 13) has 7 activation layers, and each layer receives $2^{10} - 2^{16}$ neurons. In next section, we will discuss more about the tradeoffs between prediction accuracy and overhead.

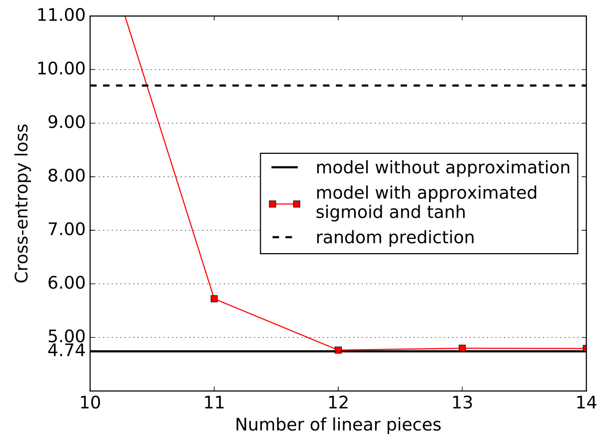

Figure 15: Cross-entropy loss for models with approximated sigmoid/tanh, evaluate over the full PTB test set.

	Latency (s)		Message Sizes (MB)		Accuracy
	offline	online	offline	online	%
ReLU/CNN/MNIST (Figure 12)	3.58	5.74	20.9	636.6	99.0
ReLU/CNN/CIFAR-10 (Figure 13)	472	72	3046	6226	81.61
Sigmoidal/LSTM/PTB (Figure 14)	13.9	4.39	86.7	474	cross-entropy loss:4.79

Table 5: Performance of MiniONN transformations of models with common activation functions and pooling operations.

7 COMPLEXITY, ACCURACY AND OVERHEAD

In Section 6, we demonstrated that, unlike prior work, MiniONN can transform existing neural networks into oblivious variants. However, by simplifying the neural network model a designer can trade off a small sacrifice in prediction accuracy with a large reduction in the overhead associated with the ONN.

The relationship between model complexity and prediction accuracy is well-known ([29], Chapter 6). In neural networks, model complexity depends on the network structure: the number of neurons (size of output from each layer), types of operations (e.g., choice of activation functions) and the number of layers in the network. While prediction accuracy can increase with model complexity, it eventually *saturates* with some level of complexity.

Model complexity vs. prediction overhead. The overhead of linear transformation is the same as non-private neural networks, since we introduce a precomputation phase to generate dot-product triples. Therefore, to investigate the overhead introduced by MiniONN, we only need to consider the activation functions and pooling operations in a given neural network model. Figure 16 shows the performance of oblivious ReLU, oblivious square, oblivious sigmoid, and oblivious max operations (used in both pooling and maxout activation functions). Both message size and latency grow sublinearly as the number of invocations increases. The experiments are

[7] http://www.fit.vutbr.cz/~imikolov/rnnlm/
[8] https://www.tensorflow.org/tutorials/recurrent, accessed April 20, 2017. We used the 'small' model configuration.
[9] This is exactly as in the Theano deep learning framework [9], where this approximation for numerical stability.

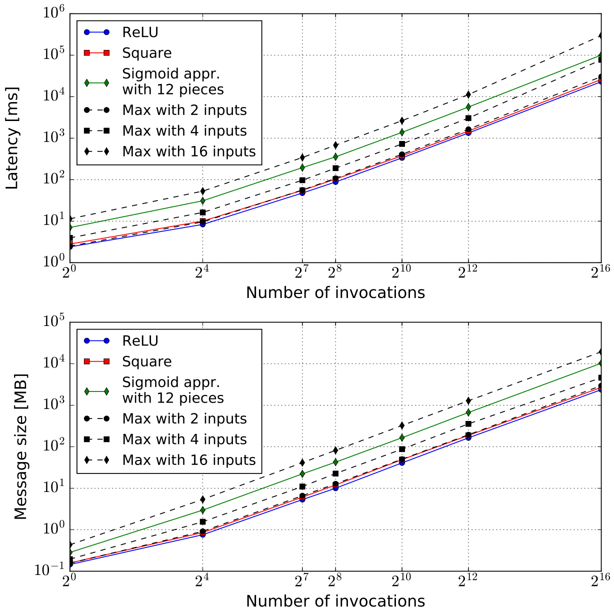

Figure 16: Overhead of oblivious activation functions.

repeated five times for each point. The standard deviation is below 2.5% for all points.

Model complexity vs. prediction accuracy. The largest contribution to overhead in the online phase are due to activation function usage. We evaluated the performance of our ReLU/CNN/MNIST network (Figure 12) by decreasing the number of neurons in linear layers and the number of channels in convolutional layers, to a fraction α of the original value, according to the changes introduced in Figure 17. This effectively reduced the number of activation function instances to the same fraction.

(1) *Convolution:* input image 28×28, window size 5×5, stride $(1, 1)$, number of output channels of $\lfloor \alpha \cdot 16 \rfloor$: $\mathbb{R}^{\lfloor \alpha \cdot 16 \rfloor \times 576} \leftarrow \mathbb{R}^{\lfloor \alpha \cdot 16 \rfloor \times 25} \cdot \mathbb{R}^{25 \times 576}$.

(4) *Convolution:* window size 5×5, stride $(1, 1)$, number of output channels $\lfloor \alpha \cdot 16 \rfloor$: $\mathbb{R}^{\lfloor \alpha \cdot 16 \rfloor \times (\lfloor \alpha \cdot 16 \rfloor)^2} \leftarrow \mathbb{R}^{\lfloor \alpha \cdot 16 \rfloor \times 400} \cdot \mathbb{R}^{400 \times (\lfloor \alpha \cdot 16 \rfloor)^2}$.

(7) *Fully Connected:* fully connects the incoming $\lfloor \alpha \cdot 16 \rfloor \cdot 16$ nodes to the outgoing $\lfloor \alpha \cdot 100 \rfloor$ nodes: $\mathbb{R}^{\lfloor \alpha \cdot 100 \rfloor \times 1} \leftarrow \mathbb{R}^{\lfloor \alpha \cdot 100 \rfloor \times (\lfloor \alpha \cdot 16 \rfloor \cdot 16)} \cdot \mathbb{R}^{(\lfloor \alpha \cdot 16 \rfloor \cdot 16) \times 1}$.

(9) *Fully Connected:* fully connects the incoming $\lfloor \alpha \cdot 100 \rfloor$ nodes to the outgoing 10 nodes: $\mathbb{R}^{10 \times 1} \leftarrow \mathbb{R}^{10 \times \lfloor \alpha \cdot 100 \rfloor} \cdot \mathbb{R}^{\lfloor \alpha \cdot 100 \rfloor \times 1}$.

Figure 17: Alternative ReLU/CNNs trained from the MNIST dataset.

Figure 18 shows how prediction accuracy varies with α. It is clear that the decline in prediction accuracy is very gradual in the range $0.25 < \alpha < 1$ (corresponding $2^{11.3}$ and $2^{13.3}$ ReLU invocations). From Figure 16, we observe that when α drops by 75% from 1 ($2^{13.3}$ invocations) to 0.25 ($2^{11.3}$ invocations), performance overhead also drops roughly by 75% (for both latency and message size) but accuracy drops only by less than a percentage point.

Accuracy vs. overhead. In Table 6, we estimated the overhead for smaller variants of the ReLU/CNN/MNIST network, w.r.t. to the base network overhead in Table 5. For example, columns 2 and 3 of

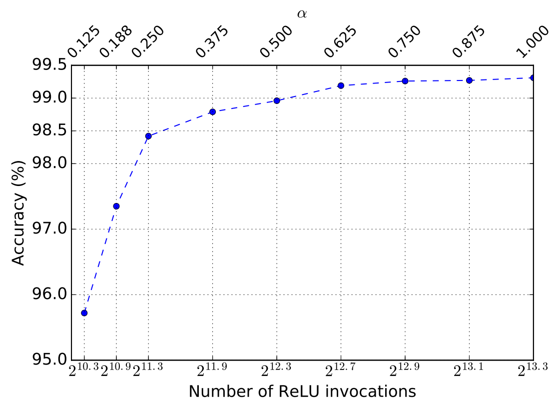

Figure 18: Model complexity vs. accuracy.

Table 6 show the estimated latencies and message sizes for different accuracy levels. Thus, if the latency and message size for a particular ONN is perceived as too high, the designer has the option of choosing a suitable point in the accuracy vs. overhead tradeoff. The overhead estimates were approximated, but reasonably accurate. For instance, an actual ReLU/CNN/MNIST model with only 25% ReLU invocations results in 1.51s latency and 159.2MB message sizes, both of which are close to the estimate for $\alpha = 0.25$ in Table 6.

8 RELATED WORK

Barni et al. [7] made the first attempt to construct oblivious neural networks. They simply have $\mathcal{S}$ do linear operations on $\mathcal{C}$'s encrypted data and send the results back to $\mathcal{C}$, who decrypts, applies the non-linear transformations on the plaintexts, and re-encrypts the results before sending them to $\mathcal{S}$ for next layer processing. Orlandi et al. [46] noticed that this process leaks significant information about $\mathcal{S}$'s neural network, and proposed a method to obscure the intermediate results. For example, when $\mathcal{S}$ needs to know $sign(x)$ from $E(pk_c, x)$, they have $\mathcal{S}$ sends $a \otimes E(pk_c, x)$ to $\mathcal{C}$ with $a > 0$. Obviously, this leaks the sign of x to $\mathcal{C}$. Our work is targeted for the same setting as these works but provides stricter security guarantees (no intermediate results will be leaked) and has significantly better performance.

Gilad-Bachrach et al. [27] proposed CryptoNets based on leveled homomorphic encryption (LHE). They introduced a simple square activation function [27]: $f(y) = y^2$, because CryptoNets cannot

α	Overhead		Accuracy (%)
	Latency (s)	Message size (MB)	
1.000	5.72*	636.6*	99.31
0.875	5.01	557.0	99.27
0.750	4.29	447.5	99.26
0.625	3.58	397.9	99.19
0.500	2.87	317.6	98.96
0.375	2.15	238.7	98.79
0.250	1.44 (1.51*)	158.4 (159.2*)	98.42
0.188	1.07	119	97.35
0.125	0.72	79.0	95.72

Table 6: Accuracy vs. overhead. * denotes actual values.

support commonly used activation functions due to the limitations of LHE. They also used mean pooling instead of max pooling for the same reason, even though the latter is more commonly used. In contrast, MiniONN supports all operations commonly used by neural network designers, does not require changes to how neural networks are trained, and has significantly lower overheads at prediction time. The privacy guarantees to the client are identical. However, while CryptoNets can hide all information about model from clients, MiniONN hides the model values (e.g., weight matrices and bias vectors) while disclosing the number of layers, sizes of weight matrices and the types of operations used in each layer. We argue that this is a justifiable tradeoff for two reasons. First, the performance gain resulting from the tradeoff are truly significant (e.g., 740-fold improvement in online latency). Second, details of a model that are disclosed by MiniONN (like the number of layers and the types of operations) are exactly those that are described in academic and white papers. Model values (like weight matrices and bias vectors in each layer) are usually not disclosed in such literature.

Chabanne et al. [15] also noticed the limited accuracy guarantees of the square function in CryptoNets. They approximated ReLU using a low degree polynomial, and added a normalization layer to make a stable and normal distributed inputs to the activation layer. However, they require a multiplicative depth of 6 in LHE, and they did not provide benchmark results in their paper.

Most of the related works focus on the privacy of training phase (see [4, 5, 30]). For example, Graepel et al. [30] proposed to use training algorithms that can be expressed as low degree polynomials, so that the training phase can be done over encrypted data. Aslett et al. [4, 5] presented ways to train both simple models (e.g., Naive Bayes) as well as more advanced models (e.g., random forests) over encrypted data. The work on differential privacy can also guarantee the privacy in training phase (see [2, 26, 50]). By leveraging Intel SGX combined with several data-oblivious algorithms, Ohrimenko et al. [45] proposed a way to enable multiple parties to jointly run a training algorithm while guaranteeing the privacy of their individual datasets.

Recently, in SecureML Mohassel and Zhang proposed a two-server model for privacy-preserving training [43]. Specifically, the data owners distribute their data among two non-colluding servers to train various models including neural networks using secure two-party computation (2PC). While their focus is on training, they also support privacy-preserving predictions. As such their work is closest to ours. Independently of us, they too use a precomputation stage to reduce the overhead during the online prediction phase, support some popular activation functions like ReLU and use approximations where necessary. MiniONN is different from their work in several ways. First, by using the SIMD batch processing technique, MiniONN achieves a significant reduction in the overhead during precomputation without affecting the online phase (Section 6.1). Second, their approximations require changes to how models are trained while the distinguishing characteristic of MiniONN is that it imposes no such requirement.

9 CONCLUSION AND FUTURE WORK

In this paper, we presented MiniONN, which is the first approach that can transform any common neural network into an oblivious form. Our benchmarks show that MiniONN achieves significantly lower response latency and message sizes compared to prior work [27, 43].

We intend to design easy-to-use interfaces that allow developers without any cryptographic background to use MiniONN directly. We also intend to investigate whether our approach is applicable to other machine learning models. As a next step, we plan to apply MiniONN to the neural networks that are being used in production.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, GA, 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[2] Martín Abadi, Andy Chu, Ian Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. 2016. Deep Learning with Differential Privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 308–318. https://doi.org/10.1145/2976749.2978318

[3] Eliana Angelini, Giacomo di Tollo, and Andrea Roli. 2008. A neural network approach for credit risk evaluation. *The quarterly review of economics and finance* 48, 4 (2008), 733–755.

[4] Louis JM Aslett, Pedro M Esperança, and Chris C Holmes. 2015. Encrypted statistical machine learning: new privacy preserving methods. *arXiv preprint arXiv:1508.06845* (2015).

[5] Louis JM Aslett, Pedro M Esperança, and Chris C Holmes. 2015. A review of homomorphic encryption and software tools for encrypted statistical machine learning. *arXiv preprint arXiv:1508.06574* (2015).

[6] Mauro Barni, Pierluigi Failla, Vladimir Kolesnikov, Riccardo Lazzeretti, Ahmad-Reza Sadeghi, and Thomas Schneider. 2009. Secure Evaluation of Private Linear Branching Programs with Medical Applications. In *Computer Security - ESORICS 2009, 14th European Symposium on Research in Computer Security, Saint-Malo, France, September 21-23, 2009. Proceedings.* 424–439. http://dx.doi.org/10.1007/978-3-642-04444-1_26

[7] M. Barni, C. Orlandi, and A. Piva. 2006. A Privacy-preserving Protocol for Neural-network-based Computation. In *Proceedings of the 8th Workshop on Multimedia and Security (MM&Sec '06)*. ACM, New York, NY, USA, 146–151. https://doi.org/10.1145/1161366.1161393

[8] Donald Beaver. 1991. Efficient Multiparty Protocols Using Circuit Randomization. In *Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1991, Proceedings (Lecture Notes in Computer Science)*, Vol. 576. Springer, 420–432. https://doi.org/10.1007/3-540-46766-1_34

[9] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. 2010. Theano: A CPU and GPU math compiler in Python. In *Proc. 9th Python in Science Conf.* 1–7.

[10] Christopher M. Bishop. 2006. *Pattern Recognition and Machine Learning (Information Science and Statistics).* Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[11] Dan Bogdanov, Roman Jagomägis, and Sven Laur. 2012. A Universal Toolkit for Cryptographically Secure Privacy-preserving Data Mining. In *Proceedings of the 2012 Pacific Asia Conference on Intelligence and Security Informatics (PAISI'12)*. Springer-Verlag, Berlin, Heidelberg, 112–126. https://doi.org/10.1007/978-3-642-30428-6_9

[12] Joppe W. Bos, Kristin Lauter, Jake Loftus, and Michael Naehrig. 2013. *Improved Security for a Ring-Based Fully Homomorphic Encryption Scheme.* Springer Berlin Heidelberg, Berlin, Heidelberg, 45–64. https://doi.org/10.1007/978-3-642-45239-0_4

[13] Raphael Bost, Raluca Ada Popa, Stephen Tu, and Shafi Goldwasser. 2015. Machine Learning Classification over Encrypted Data. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015.* http://www.internetsociety.org/doc/machine-learning-classification-over-encrypted-data

[14] Justin Brickell, Donald E. Porter, Vitaly Shmatikov, and Emmett Witchel. 2007. Privacy-preserving remote diagnostics. In *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*. 498–507. http://doi.acm.org/10.1145/1315245.1315307

[15] Hervé Chabanne, Amaury de Wargny, Jonathan Milgram, Constance Morel, and Emmanuel Prouff. 2017. Privacy-Preserving Classification on Deep Neural Network. Cryptology ePrint Archive, Report 2017/035. (2017). http://eprint.iacr.org/2017/035.

[16] Jia-Ren Chang and Yong-Sheng Chen. 2015. Batch-normalized maxout network in network. *arXiv preprint arXiv:1511.02583* (2015).

[17] Kumar Chellapilla, Sidd Puri, and Patrice Simard. 2006. High performance convolutional neural networks for document processing. In *Tenth International Workshop on Frontiers in Handwriting Recognition*. Suvisoft.

[18] Dan Ciregan, Ueli Meier, and Jürgen Schmidhuber. 2012. Multi-column deep neural networks for image classification. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*. IEEE, 3642–3649.

[19] G. E. Dahl, D. Yu, L. Deng, and A. Acero. 2012. Context-Dependent Pre-Trained Deep Neural Networks for Large-Vocabulary Speech Recognition. *IEEE Transactions on Audio, Speech, and Language Processing* 20, 1 (Jan 2012), 30–42. https://doi.org/10.1109/TASL.2011.2134090

[20] Daniel Demmler, Thomas Schneider, and Michael Zohner. 2015. ABY-A Framework for Efficient Mixed-Protocol Secure Two-Party Computation.. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*.

[21] Paul Dierckx. 1995. *Curve and surface fitting with splines*. Oxford University Press.

[22] Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. 2015. Manual for using homomorphic encryption for bioinformatics. *Microsoft Research* (2015).

[23] Taher ElGamal. 1985. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. In *CRYPTO (LNCS)*, Vol. 196. Springer, 10–18. https://doi.org/10.1007/3-540-39568-7_2

[24] Rasool Fakoor, Faisal Ladhak, Azade Nazi, and Manfred Huber. 2013. Using deep learning to enhance cancer diagnosis and classification. In *Proceedings of the International Conference on Machine Learning*.

[25] Matthew Fredrikson, Eric Lantz, Somesh Jha, Simon Lin, David Page, and Thomas Ristenpart. 2014. Privacy in Pharmacogenetics: An End-to-End Case Study of Personalized Warfarin Dosing. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, 17–32. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/fredrikson_matthew

[26] Arik Friedman and Assaf Schuster. 2010. Data Mining with Differential Privacy. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '10)*. ACM, New York, NY, USA, 493–502. https://doi.org/10.1145/1835804.1835868

[27] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. 2016. CryptoNets: Applying neural networks to encrypted data with high throughput and accuracy. In *Proceedings of The 33rd International Conference on Machine Learning*. 201–210.

[28] O. Goldreich, S. Micali, and A. Wigderson. 1987. How to Play ANY Mental Game. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing (STOC '87)*. ACM, New York, NY, USA, 218–229. https://doi.org/10.1145/28395.28420

[29] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[30] Thore Graepel, Kristin E. Lauter, and Michael Naehrig. 2012. ML Confidential: Machine Learning on Encrypted Data. In *Information Security and Cryptology - ICISC 2012 - 15th International Conference, Seoul, Korea, November 28-30, 2012, Revised Selected Papers*. 1–21. http://dx.doi.org/10.1007/978-3-642-37682-5_1

[31] Benjamin Graham. 2014. Fractional max-pooling. *arXiv preprint arXiv:1412.6071* (2014).

[32] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[33] Eric Jones, Travis Oliphant, P Peterson, et al. 2001. SciPy: Open source scientific tools for Python. (2001).

[34] Nicola Jones. 2014. Nature: Computer science: The learning machines. (2014). http://www.nature.com/news/computer-science-the-learning-machines-1.14481.

[35] Alex Krizhevsky and Geoffrey Hinton. 2009. Learning multiple layers of features from tiny images. (2009). http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.222.9220&rep=rep1&type=pdf.

[36] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 1097–1105. http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[37] Yann LeCun, Corinna Cortes, and Christopher JC Burges. 1998. The MNIST database of handwritten digits. (1998). http://yann.lecun.com/exdb/mnist/.

[38] Chen-Yu Lee, Patrick W. Gallagher, and Zhuowen Tu. 2016. Generalizing Pooling Functions in Convolutional Neural Networks: Mixed, Gated, and Tree. In *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics, AISTATS 2016, Cadiz, Spain, May 9-11, 2016*. 464–472. http://jmlr.org/proceedings/papers/v51/lee16a.html

[39] Dong C Liu and Jorge Nocedal. 1989. On the limited memory BFGS method for large scale optimization. *Mathematical programming* 45, 1 (1989), 503–528.

[40] Mitchell P Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. 1993. Building a large annotated corpus of English: The Penn Treebank. *Computational linguistics* 19, 2 (1993), 313–330.

[41] Tomáš Mikolov, Ilya Sutskever, Anoop Deoras, Hai-Son Le, Stefan Kombrink, and Jan Cernocky. 2012. Subword language modeling with neural networks. *preprint (http://www. fit. vutbr. cz/imikolov/rnnlm/char. pdf)* (2012).

[42] Dmytro Mishkin and Jiri Matas. 2015. All you need is a good init. *arXiv preprint arXiv:1511.06422* (2015).

[43] Payman Mohassel and Yupeng Zhang. 2017. SecureML: A System for Scalable Privacy-Preserving Machine Learning. Cryptology ePrint Archive, Report 2017/396. (May 2017). https://eprint.iacr.org/2017/396.

[44] Kevin P Murphy. 2012. *Machine learning: a probabilistic perspective*. MIT press.

[45] Olga Ohrimenko, Felix Schuster, Cedric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. 2016. Oblivious Multi-Party Machine Learning on Trusted Processors. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 619–636. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/ohrimenko

[46] C. Orlandi, A. Piva, and M. Barni. 2007. Oblivious Neural Network Computing via Homomorphic Encryption. *EURASIP J. Inf. Secur.* 2007, Article 18 (Jan. 2007), 10 pages. https://doi.org/10.1155/2007/37343

[47] Pascal Paillier. 1999. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *EUROCRYPT (LNCS)*, Jacques Stern (Ed.), Vol. 1592. Springer, 223–238. https://doi.org/10.1007/3-540-48910-X_16

[48] Pille Pullonen and Sander Siim. 2015. Combining Secret Sharing and Garbled Circuits for Efficient Private IEEE 754 Floating-Point Computations. In *Financial Cryptography and Data Security - FC 2015 International Workshops, BITCOIN, WAHC, and Wearable, San Juan, Puerto Rico, January 30, 2015, Revised Selected Papers*. 172–183. https://doi.org/10.1007/978-3-662-48051-9_13

[49] Ikuro Sato, Hiroki Nishimura, and Kensuke Yokoi. 2015. Apac: Augmented pattern classification with neural networks. *arXiv preprint arXiv:1505.03229* (2015).

[50] Reza Shokri and Vitaly Shmatikov. 2015. Privacy-Preserving Deep Learning. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, New York, NY, USA, 1310–1321. https://doi.org/10.1145/2810103.2813687

[51] Reza Shokri, Marco Stronati, and Vitaly Shmatikov. 2016. Membership inference attacks against machine learning models. *arXiv preprint arXiv:1610.05820* (2016).

[52] N. P. Smart and F. Vercauteren. 2014. Fully homomorphic SIMD operations. *Designs, Codes and Cryptography* 71, 1 (2014), 57–81. https://doi.org/10.1007/s10623-012-9720-4

[53] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. 2014. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806* (2014).

[54] Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2016. Stealing Machine Learning Models via Prediction APIs. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 601–618. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/tramer

[55] Li Wan, Matthew Zeiler, Sixin Zhang, Yann L. Cun, and Rob Fergus. 2013. Regularization of Neural Networks using DropConnect. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, Sanjoy Dasgupta and David Mcallester (Eds.). JMLR Workshop and Conference Proceedings, 1058–1066. http://jmlr.org/proceedings/papers/v28/wan13.pdf

[56] David J. Wu, Tony Feng, Michael Naehrig, and Kristin E. Lauter. 2016. Privately Evaluating Decision Trees and Random Forests. *Privacy Enhancing Technologies (PoPETs)* 2016, 4 (2016), 335–355. http://dx.doi.org/10.1515/popets-2016-0043

[57] Andrew Chi-Chih Yao. 1982. Protocols for Secure Computations (Extended Abstract). In *Foundations of Computer Science (FOCS'82)*. IEEE, 160–164.

[58] Andrew C.-C. Yao. 1986. How to Generate and Exchange Secrets. In *Foundations of Computer Science (FOCS'86)*. IEEE, 162–167.

[59] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. 2014. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329* (2014).